5 Cool Things About Rust

Contents

Introduction

Rust is a safe systems language designed by Mozilla as an alternative to C and C++. It is already being used in the creation of large software systems, including its own compiler (rustc), and a web browser rendering engine called Servo.

Rust focuses on providing language features that guarantee the nonexistence of certain classes of common errors, without impairing program performance. Those errors include:

- Memory leaks
- Data races
- Buffer overflows
- Segmentation faults

In this small essay I will walk through a collection of five Rust features and tools that make creating systems in Rust a joy. They are: Cargo (the Rust build tool and package manager), immutable-by-default variables, the trait system, the borrow checker, and pattern matching.

Cargo

Cargo is Rust's build tool and package manager, and while it's still fairly new it is already quite good at its job. To start a new library in Rust, simply type:

```
cargo new <name>
```

If you want your project to build into a binary instead, type:

```
cargo new <name> --bin
```

In either case, Cargo will automatically generate your necessary file structure, setup a Git repo, and create a skeleton configuration file (`Cargo.toml`) to enable you to build your code right away.

Cargo also gives handy commands to build and run your code:

```
cargo build
cargo run
```

Both commands will automatically fetch the dependencies defined in the `Cargo.toml` file and build your code with them, and the second one will then automatically execute the generated binary (if you didn't make a binary repo, then this is all moot).

If you want to make sure that dependency versions are kept consistent between contributors to a project, just use Cargo's auto-generated `Cargo.lock` file. The first time a build succeeds, Cargo generates this file to keep track of which version each dependency was on. Then in future builds it will continue to use that exact version until it is explicitly told to update via:

```
cargo update <dependency name>
```

This way, all contributors on a project are guaranteed to build using the exact same versions of all dependencies, and changes in versions are saved in the Git repository's history (making them easy to track).

Cargo also has a nice system for running tests. With the following command, Cargo will automatically compile your code with tests included, and then run all the tests you've defined and report on their success:

```
cargo test
```

Rust itself has language-level support for testing, and Cargo makes using those tests you've defined extremely easy.

For all of these reasons, Cargo is an excellent part of the Rust ecosystem, and one which any real-world Rust project would be wrong to do without.

Mutability

In most languages, values are mutable by default. For example, in C++, the following is totally valid:

```
int x = 5;
x = 10;
```

If you wanted to make the above invalid, you would have to declare `x` to be `const`, thus making it immutable:

```
const int x = 5;
x = 10;  // Error! This isn't allowed.
```

So, C++ is mutable-by-default. For small programs this is often fine, and it has been the standard operating procedure in popular languages for decades. However, it has the negative effect of making it easy to accidentally mutate something you didn't expect to be changed. After all, when you have to explicitly say that something is immutable, it is easy to forget.

Rust avoids this by making everything immutable-by-default. So in Rust, the following is invalid:

```
let x: int = 5;
x = 10;  // Error! This isn't allowed.
```

If you want to be able to mutate `x`, you have to use the keyword `mut`.

```
let mut x: int = 5;
x = 10;
```

This may seem like a pain, but it actually makes life easier in a number of different ways. First, it drastically reduces the likelihood that some varaible you've declared will change values unexpectedly during runtime. It also makes tracking down variables that *can* change a lot easier (just look for the ones

declared using `mut`). But the most compelling reason for this decision has to do with concurrency.

One of the major problems in concurrency is the issue of *shared state*. When values can modified by two threads running simultaneously it is very easy to create *data races*, where the result is based on the order in which the threads complete.

When variables are mutable-by-default, this can happen quite easily. If even a single variable is modified in two threads, the possibly of data races arises.

Yet if variables are immutable-by-default, it becomes much easier to avoid the possibility of sharing state between threads. Even better, it becomes easier for the compiler to keep track of whether variables shared between threads are mutable or immutable, and provide errors and warnings as necessary.

Rust goes even further by providing extremely good error messages related to mutability mistakes. Take for example the incorrect Rust code from earlier:

```
let x: int = 5;
x = 10;
```

Here is what the compiler has to say about that:

```
main.rs:3:3: 3:9 error: re-assignment of immutable variable `x`
main.rs:3   x = 10;
            ^~~~~~
main.rs:2:7: 2:8 note: prior assignment occurs here
main.rs:2   let x: int = 5;
                ^
error: aborting due to previous error
```

The compiler noticed immediately that I was trying to reassign an immutable variable, and told me both where the reasignment happens and where the original happened.

Traits

Rust is not an Object-Oriented programming language. It has no notion of classes or objects, nor a notion of inheritance. Instead, Rust has the **trait system**.

Rust's trait system allows for the creation of generic functions, which can take as input any type which implements the desired trait. Take for example the following function:

```
fn add_one(x: int) -> int {
    x + 1
}
```

This is a function that takes in an integer, adds 1 to it, and then returns the result. It's fairly straightforward, but only works for variables of type `int`. If you wanted to use it on something else, you would have to define an equivalent function for the other type. This is what happens in C. In Rust, you can instead write a function using the trait system to accept anything that implements the proper trait.

```
fn add_one<T: Add>(x: T) -> T {
    x + 1
}
```

In this case, the function accepts anything of some type `T` that implements the `Add` trait. Internally, the Add trait defines a function like so:

```
pub trait Add<RHS, Result> {
    fn add(&self, rhs: &RHS) -> Result;
}
```

Where `RHS` is the type of the thing being added, the `Result` is the type of the result of the addition. So anything that wants to allow adding simply has to implement the add trait like this:

```
struct Foo;

impl Add<Foo, Foo> for Foo {
    fn add(&self, _rhs: &Foo) -> Foo {
      println!("Adding!");
      *self
  }
}

fn main() {
  Foo + Foo;
}
```

In this case, the code will print `Adding!` once, as the `add()` function is called once by the `+` operator.

One thing to note is that as of now, Rust's trait system is not as powerful as the C++ template system (which is actually a Turing complete metaprogramming

language). However, it is sufficiently expressive for what is needs to do, and provides a strong basis for the creation and usage of generic functions.

(Incidentally, the Rust trait system borrows heavily from the Haskell typeclass system, and is also inspired by C++'s once-proposed "concepts" system, which was never implemented in the language.)

The Borrow Checker

There is a concept in C++ (and other languages too, although it started in C++) called RAII. It stands for "Resource Acquisition Is Initialization" and exists to allow for the exception safe construction and destruction of objects.

Essentially, RAII says that an object is constructed when it is acquired, and destructed when nothing owns it anymore. This way memory isn't leaked because someone forgot to deallocate. It all happens automatically.

This is codified in Rust as the *lifetime system*. Rust tracks how long everything in the language exists, and will automatically allocate and initialize values when they are assigned to variables, and automatically deallocate them when their owner goes out of scope, passing along ownership as necessary in the meantime.

This means that Rust has to be very careful about tracking when variables own values, which is why the **Borrow Checker** exists. Let's look at some code:

```
fn dangling() -> &int {
    let i = 1234;
    return &i;
}

fn add_one() -> int {
    let num = dangling();
    return *num + 1;
}

fn main() {
    add_one();
}
```

This code fails to compile with the following error message:

```
temp.rs:3:11: 3:13 error: borrowed value does not live long enough
temp.rs:3     return &i;

temp.rs:1:22: 4:1 note: borrowed pointer must be valid for
```

```
                        the anonymous lifetime #1 defined
                        on the block at 1:22...
temp.rs:1 fn dangling() -> &int {
temp.rs:2     let i = 1234;
temp.rs:3     return &i;
temp.rs:4 }

temp.rs:1:22: 4:1 note: ...but borrowed value is
                        only valid for the block at 1:22
temp.rs:1 fn dangling() -> &int {
temp.rs:2     let i = 1234;
temp.rs:3     return &i;
temp.rs:4  }
error: aborting due to previous error
```

What this means is that the value `1234` created inside the `dangling()` function only exists until the end of the function, when it is automatically destroyed. So when you try to give out a reference to it, you're giving out a reference to something that no longer exists. In other language (like C++) this example will lead to the dereferencing of uninitialized memory, which is a security hole. In Rust, the borrow checker sees that you're handing out a dead reference, and stops you from doing it.

But what if you really wanted to do it this way? Well, with a small change to the code you can! Here it is:

```
fn dangling() -> Box<int> {
    let i = box 1234;
    return i;
}

fn add_one() -> int {
    let num = dangling();
    return *num + 1;
}

fn main() {
    add_one();
}
```

In this case, the `dangling()` function was modified to use `box`. This means that `i` is now a unique pointer to the memory location containing `1234`. When the function terminates, it transfers ownership of that pointer to the caller (in this case `num` inside the `add_one()` function). When `add_one()` terminates, `num` goes out of scope, and the memory it points to is reclaimed.

This isn't the only thing that the borrow checker can do. Let's take a look at another case:

```
fn main() {
  let mut x = 5i;
  let y = &mut x;
  let z = &mut x;
}
```

This code will fail to compile with the following message:

```
main.rs:4:16: 4:17 error: cannot borrow `x` as mutable
                          more than once at a time
main.rs:4   let z = &mut x;
                        ^
main.rs:3:16: 3:17 note: previous borrow of `x` occurs here;
                         the mutable borrow prevents subsequent
                         moves, borrows, or modification of `x`
                         until the borrow ends
main.rs:3   let y = &mut x;
                        ^
main.rs:5:2: 5:2 note: previous borrow ends here
main.rs:1 fn main() {
main.rs:2   let mut x = 5i;
main.rs:3   let y = &mut x;
main.rs:4   let z = &mut x;
main.rs:5 }
          ^
error: aborting due to previous error
```

Here, Rust is saying that we can't have more than one mutable reference to the same data at the same time. This makes sense. Thinking back to the shared state problem, two mutable references is just as bad as having two pointers. But what if we changed them to normal references?

```
fn main() {
  let mut x = 5i;
  let y = &x;
  let z = &x;
}
```

It compiles just fine! Essentially, borrowing (which is what a reference is) is fine, so long as the thing you're borrowing lives long enough. But owning (which is what mutable references and pointer assignment do) has some restrictions to make sure you avoid shared state and other sticky problems. And the Borrow Checker double checks all of those rules for you and tells you if you're breaking them!

Pattern Matching

In C, if you want to allocate some memory, you use a function like `malloc()`. This allocates the memory, or returns a NULL pointer if it fails. In order to make sure you never dereference a null pointer, you therefore have to check if its null after calling `malloc()`. And you have to remember to this every single time you allocate something. If you don't, you have a security hole.

What you really want is some way to represent the idea that the function failed, a way to safely represent the concept of nothing. In Rust, this is done through an enum called Option, and a concept called **Pattern Matching**.

Let's take a look at an example:

```
use std::rand;

fn odd_or_none () -> Option<uint> {
  let x = rand::random::<uint>() % 100u;
  if x % 2 == 1 {
    Some(x)
  }
  else {
    None
  }
}

fn main () {
  let x: Option<uint> = odd_or_none();

  match x {
    Some(x) => println!("Odd: {}", x),
    None    => println!("Nothing!")
  }
}
```

In this program, `odd_or_none()` generates a random number. If that number is odd, it returns the number, otherwise it returns None. When you want to use the number it returns, you pattern matching against the return value and extract it.

So what happens if you forgot to check for `None`?

```
use std::rand;

fn odd_or_none () -> Option<uint> {
  let x = rand::random::<uint>() % 100u;
```

```
  if x % 2 == 1 {
    Some(x)
  }
  else {
    None
  }
}

fn main () {
  let x: Option<uint> = odd_or_none();

  match x {
    Some(x) => println!("Odd: {}", x)
  }
}
```

You get the error message:

```
main.rs:16:3: 18:4 error: non-exhaustive patterns:
                          `None` not covered [E0004]
main.rs:16   match x {
main.rs:17     Some(x) => println!("Odd: {}", x)
main.rs:18   }
error: aborting due to previous error
```

Look at that! The compiler noticed that you didn't check for `None` and refused to compile the program. This is because in Rust, pattern matching has to cover every possible case. So the compiler stops you from ever forgetting to check for NULL (or `None` in this case).

Let's look at another example, this time it's a simple guess-the-number game:

```
use std::io;
use std::rand;

fn main() {
    println!("Guess the number!");

    let secret_number = (rand::random::<uint>() % 100u) + 1u;

    loop {

        println!("Please input your guess.");

        let input = io::stdin().read_line()
```

```
                            .ok()
                            .expect("Failed to read line");
        let input_num: Option<uint> =
          from_str(input.as_slice().trim());

        let num = match input_num {
            Some(num) => num,
            None      => {
                println!("Please input a number!");
                continue;
            }
        };


        println!("You guessed: {}", num);

        match cmp(num, secret_number) {
            Less    => println!("Too small!"),
            Greater => println!("Too big!"),
            Equal   => {
                println!("You win!");
                return;
            },
        }
    }
}

fn cmp(a: uint, b: uint) -> Ordering {
    if a < b { Less }
    else if a > b { Greater }
    else { Equal }
}
```

This code is a bit longer than what we've seen before, but it shows us several examples of pattern matching. First you attempt to convert the input value to a number, and then pattern match on the result. If it succeeded you grab the number. If it fails, you loop again and prompt the user for another piece of input.

Then you check the input guess against the actual number. If it's smaller, you say it's too small. If it's bigger, you say it's too big. It it's equal you tell the user they've won and then quit the game. And if you forgot to check any of those cases the compiler would notice and yell at you to fix it.

This is what Pattern Matching is all about. It provides a way for the compiler to automatically verify that you're covering all of your bases, and makes it easy

for you to know if you're not. Because when you forget, the compiler tells you to add what you missed.

Conclusion

These are just a few great features in Rust. I highly encourage you to check out the rest of the language, particularly as it approaches a stable 1.0 version release (to be done sometime later this year).

Rust is an interesting language with a serious focus on facilitating the development of stable, safe, and scalable systems. Its performance is already competitive with C++ in a variety of benchmarks, and it is already receiving serious interest from a variety of real-world businesses.

If you want to learn more about Rust, go to http://rust-lang.org